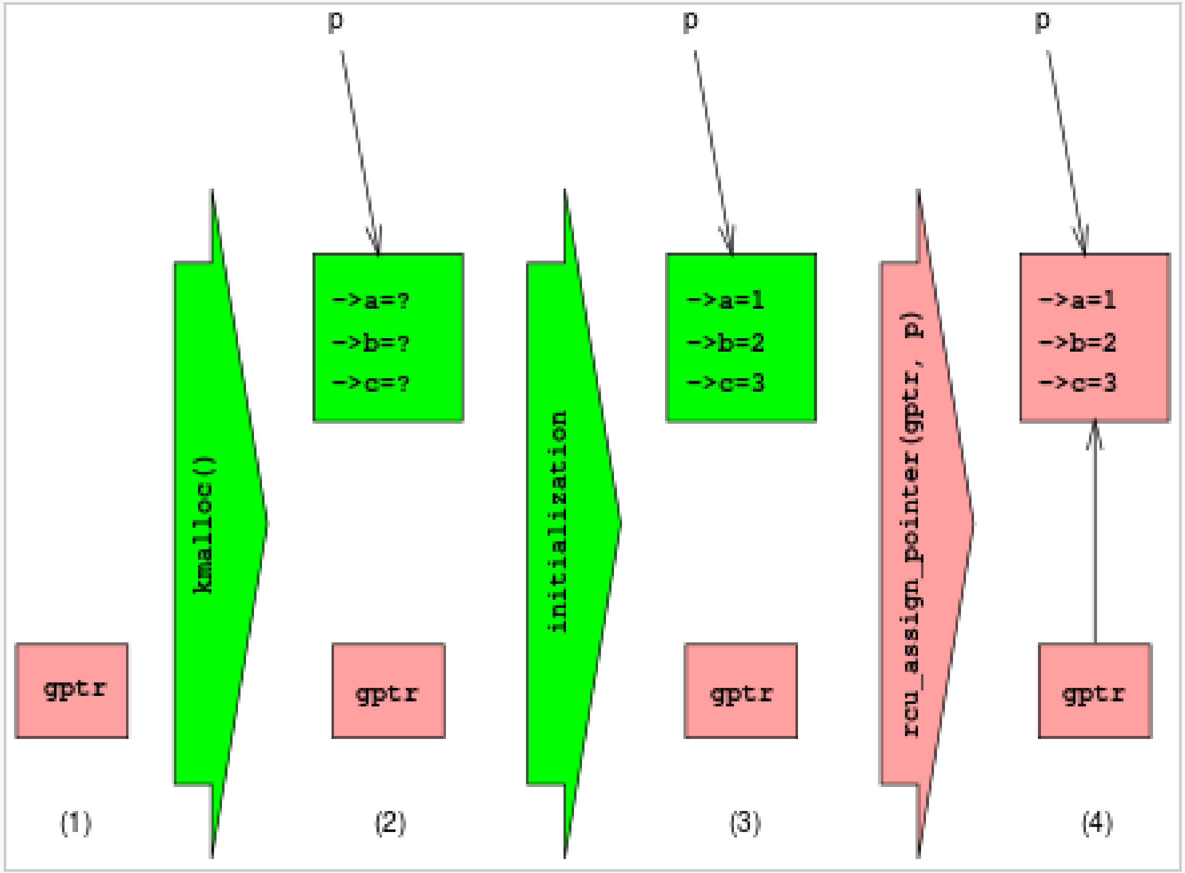
Read-copy-update

From Wikipedia, the free encyclopedia

In computer operating systems, **read-copy-update** (**RCU**) is a synchronization mechanism implementing a kind of mutual exclusion[note 1] that can sometimes be used as an alternative to a readers-writer lock. It allows extremely low overhead, wait-free reads. However, RCU updates can be expensive, as they must leave the old versions of the data structure in place to accommodate pre-existing readers. These old versions are reclaimed after all pre-existing readers finish their accesses.

Contents [hide]

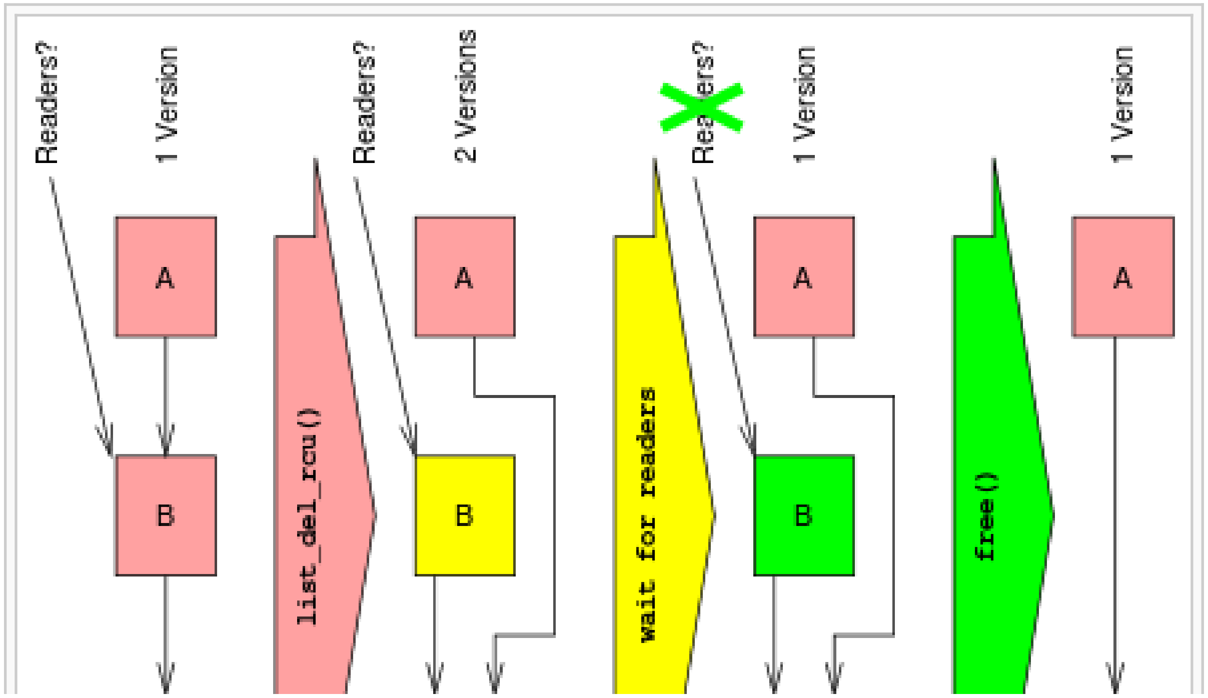
Overview [edit]



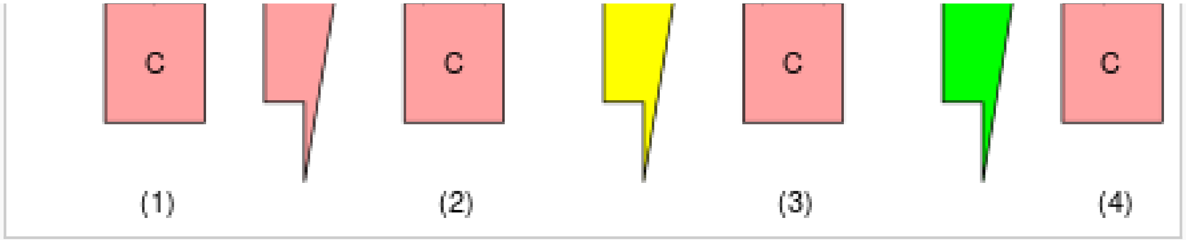
Read-copy-update insertion procedure

A key property of RCU is that readers can access a data structure even when it is in the process of being updated: There is absolutely nothing that RCU updaters can do to block readers or even to force them to retry their accesses. Some people find this property to be surprising and even counter-intuitive, but this property is absolutely essential to RCU. This overview therefore starts by showing how data can be safely inserted into and deleted from linked structures despite concurrent readers. The first diagram on the right depicts a four-state insertion procedure, with time advancing from left to right.

The first state shows a global pointer named "gptr" that is initially NULL, colored red to indicate that it might be accessed by a reader at any time, thus requiring updaters to take care. Allocating memory for a new structure transitions to the second state. This structure has indeterminate state (indicated by the question marks) but is inaccessible to readers (indicated by the green color). Because the structure is inaccessible to readers, the updater may carry out any desired operation without fear of disrupting concurrent readers. Initializing this new structure transitions to the third state, which shows the initialized values of the structure's fields. Assigning a reference to this new structure to gptr transitions to the fourth and final state. In this state, the structure is accessible to readers, and is therefore colored red. The rcu_assign_pointer() primitive is used to carry out this assignment, and ensures that the assignment is atomic in the sense that concurrent readers will either see a NULL pointer or a valid pointer to the new structure, but not some mash-up of the two values. Additional properties of rcu_assign_pointer() are described later in this article.



This procedure demonstrates how new data may be inserted into a linked data structure even though readers are concurrently traversing the data structure before, during, and after the insertion. The second diagram on the right depicts a four-state deletion procedure, again with time advancing from left to right.



Read-copy-update deletion procedure

The first state shows a linked list containing elements A, B, and C. All three elements are colored red to indicate that an RCU reader might reference any of them at any time. Using list_del_rcu() to remove element B from this list transitions to the second state. Note that the link from element B to C is left intact in order to allow readers currently referencing element B to traverse the remainder of the list. Readers accessing the link from element A will either obtain a reference to element B or element C, but either way, each reader will see a valid and correctly formatted linked list. Element B is now colored yellow to indicate that while pre-existing readers might still have a reference to element B, new readers have no way to obtain a reference. A wait-for-readers operation transitions to the third state. Note that this wait-for-readers operation need only wait for pre-existing readers, but not new readers. Element B is now colored green to indicate that readers can no longer be referencing it. Therefore, it is now safe for the updater to free element B, thus transitioning to the fourth and final state.

It is important to reiterate that in the second state different readers can see two different versions of the list, either with or without element B. In other words, RCU provides coordination in space (different versions of the list) as well as in time (different states in the deletion procedures). This is in stark contrast with more traditional synchronization primitives such as locking or transactions that coordinate in time, but not in space. RCU's use of both space and time allows exceedingly fast and scalable readers.

This procedure demonstrates how old data may be removed from a linked data structure even though readers are concurrently traversing the data structure before, during, and after the deletion. Given insertion and deletion, a wide variety of data structures can be implemented using RCU.

RCU's readers execute within *read-side critical sections*, which are normally delimited by rcu_read_lock() and rcu_read_unlock(). Any statement that is not within an RCU read-side critical section is said to be in a *quiescent state*, and such statements are not permitted to hold references to RCU-protected data structures, nor is the wait-for-readers operation required to wait for threads in quiescent states. Any time period during which each thread resides at least once in a quiescent state is called a *grace period*. By definition, any RCU read-side critical section in existence at the beginning of a given grace period must complete before the end of that grace period, which constitutes the fundamental guarantee provided by RCU. In addition, the wait-for-readers operation must wait for at least one grace period to elapse. It turns out that this guarantee can be provided with extremely small read-side overheads, in fact, in the limiting case that is actually realized by server-class Linux-kernel builds, the read-side overhead is exactly zero.[1]

RCU's fundamental guarantee may be used by splitting updates into *removal* and *reclamation* phases. The removal phase removes references to data items within a data structure (possibly by replacing them with references to new versions of these data items), and can run concurrently with RCU read-side critical sections. The reason that it is safe to run the removal phase concurrently with RCU readers is the semantics of modern CPUs guarantee that readers will see either the old or the new version of the data structure rather than a partially updated reference. Once a grace period has elapsed, there can no longer be any readers referencing the old version, so it is then safe for the reclamation phase to free (*reclaim*) the data items that made up that old version.[2]

Splitting an update into removal and reclamation phases allows the updater to perform the removal phase immediately, and to defer the reclamation phase until all readers active during the removal phase have completed, in other words, until a grace period has elapsed.[note 2]

So the typical RCU update sequence goes something like the following:[3]

1. Ensure that all readers accessing RCU-protected data structures carry out their references from within an RCU read-side critical section.
2. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.
3. Wait for a grace period to elapse, so that all previous readers (which might still have pointers to the data structure removed in the prior step) will have completed their RCU read-side critical sections.
4. At this point, there cannot be any readers still holding references to the data structure, so it now may safely be reclaimed (e.g., freed).[note 3]

In the above procedure (which matches the earlier diagram), the updater is performing both the removal and the reclamation step, but it is often helpful for an entirely different thread to do the reclamation. Reference counting can be used to let the reader perform removal so, even if the same thread performs both the update step (step (2) above) and the reclamation step (step (4) above), it is often helpful to think of them separately.

Uses [edit]

As of early 2008, there were almost 2,000 uses of the RCU API within the Linux kernel[4] including the networking protocol stacks[5] and the memory-management system.[6] As of March 2014, there were more than 9,000 uses.[7] Since 2006, researchers have applied RCU and similar techniques to a number of problems, including management of metadata used in dynamic analysis,[8] managing the lifetime of clustered objects,[9] managing object lifetime in the K42 research operating system,[10][11] and optimizing software transactional memory implementations.[12][13] Dragonfly BSD uses a technique similar to RCU that most closely resembles Linux's Sleepable

RCU (SRCU) implementation.

Advantages and disadvantages [edit]

The ability to wait until all readers are done allows RCU readers to use much lighter-weight synchronization—in some cases, absolutely no synchronization at all. In contrast, in more conventional lock-based schemes, readers must use heavy-weight synchronization in order to prevent an updater from deleting the data structure out from under them. This is because lock-based updaters typically update data items in place, and must therefore exclude readers. In contrast, RCU-based updaters typically take advantage of the fact that writes to single aligned pointers are atomic on modern CPUs, allowing atomic insertion, removal, and replacement of data items in a linked structure without disrupting readers. Concurrent RCU readers can then continue accessing the old versions, and can dispense with the atomic read-modify-write instructions, memory barriers, and cache misses that are so expensive on modern SMP computer systems, even in absence of lock contention.[14][15] The lightweight nature of RCU's read-side primitives provides additional advantages beyond excellent performance, scalability, and real-time response. For example, they provide immunity to most deadlock and livelock conditions.[note 4]

Of course, RCU also has disadvantages. For example, RCU is a specialized technique that works best in situations with mostly reads and few updates, but is often less applicable to update-only workloads. For another example, although the fact that RCU readers and updaters may execute concurrently is what enables the lightweight nature of RCU's read-side primitives, some algorithms may not be amenable to read/update concurrency.

Despite well over a decade of experience with RCU, the exact extent of its applicability is still a research topic.

Patents [edit]

The technique is covered by U.S. software patent 5,442,758, issued August 15, 1995 and assigned to Sequent Computer Systems, as well as by 5,608,893, 5,727,209, 6,219,690, and 6,886,162. The now-expired US Patent 4,809,168 covers a closely related technique. RCU is also the topic of one claim in the SCO v. IBM lawsuit.

Sample RCU interface [edit]

RCU is available in a number of operating systems, and was added to the Linux kernel in October 2002. User-level implementations such as liburcu are also available.[16]

The implementation of RCU in version 2.6 of the Linux kernel is among the better-known RCU implementations, and will be used as an inspiration for the RCU API in the remainder of this article. The core API (Application Programming Interface) is quite small:[17]

- rcu_read_lock(): Marks an RCU-protected data structure so that it won't be reclaimed for the full duration of that critical section.
- rcu_read_unlock(): Used by a reader to inform the reclaimer that the reader is exiting an RCU read-side critical section. Note that RCU read-side critical sections may be nested and/or overlapping.
- synchronize_rcu(): It blocks until all pre-existing RCU read-side critical sections on all CPUs have completed. Note that `synchronize_rcu` will *not* necessarily wait for any subsequent RCU read-side critical sections to complete. For example, consider the following sequence of events:

```
                CPU 0                  CPU 1                 CPU 2
```

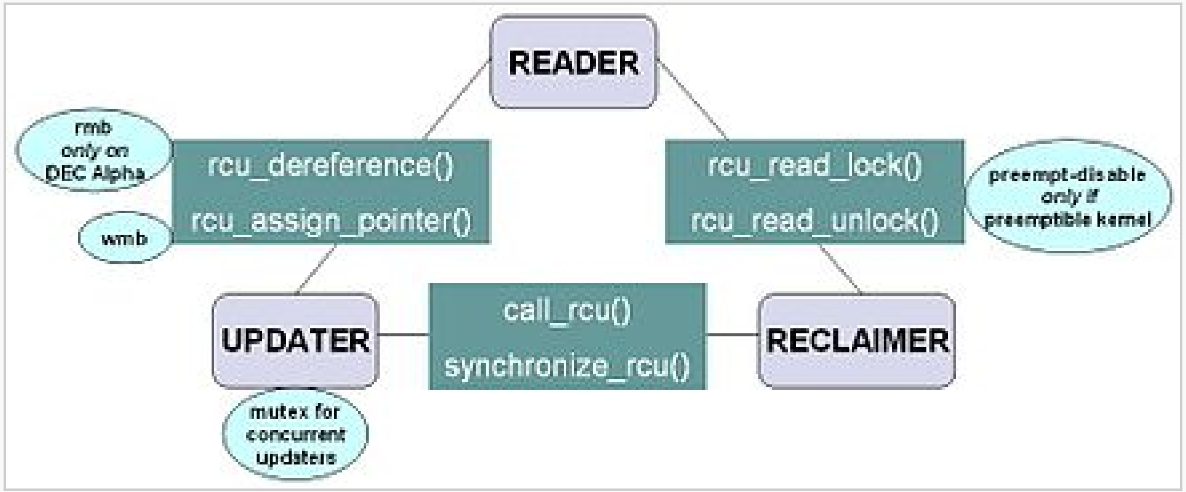
```
                 ----------------- ------------------------- ---------------
             1.  rcu_read_lock()
             2.                    enters synchronize_rcu()
             3.                                                 rcu_read_lock()
             4.  rcu_read_unlock()
             5.                     exits synchronize_rcu()
             6.
rcu_read_unlock()
```

Since `synchronize_rcu` is the API that must figure out when readers are done, its implementation is key to RCU. For RCU to be useful in all but the most read-intensive situations, `synchronize_rcu`'s overhead must also be quite small.

Alternatively, instead of blocking, synchronize_rcu may register a callback to be invoked after all ongoing RCU read-side critical sections have completed. This callback variant is called `call_rcu` in the Linux kernel.

- rcu_assign_pointer(): The updater uses this function to assign a new value to an RCU-protected pointer, in order to safely communicate the change in value from the updater to the reader. This function returns the new value, and also executes any memory barrier instructions required for a given CPU architecture. Perhaps more importantly, it serves to document which pointers are protected by RCU.
- rcu_dereference(): The reader uses `rcu_dereference` to fetch an RCU-protected pointer, which returns a value that may then be safely dereferenced. It also executes any directives required by the compiler or the CPU, for example, a volatile cast for gcc, a memory_order_consume load for C/C++11 or the memory-barrier instruction required by the old DEC Alpha CPU. The value returned by `rcu_dereference` is valid only within the

enclosing RCU read-side critical section. As with `rcu_assign_pointer`, an important function of `rcu_dereference` is to document which pointers are protected by RCU.



RCU API communications between the reader, updater, and reclaimer

The diagram on the right shows how each API communicates among the reader, updater, and reclaimer.

The RCU infrastructure observes the time sequence of `rcu_read_lock`, `rcu_read_unlock`, `synchronize_rcu`, and `call_rcu` invocations in order to determine when (1) `synchronize_rcu` invocations may return to their callers and (2) `call_rcu` callbacks may be invoked. Efficient implementations of the RCU infrastructure make heavy use of batching in order to amortize their overhead over many uses of the corresponding APIs.

Simple implementation [edit]

RCU has extremely simple "toy" implementations that can aid understanding of RCU. This section presents one such "toy" implementation that works in a non-preemptive environment.[18]

```
void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }

void call_rcu(void (*callback) (void *), void *arg)
{
        // add callback/arg pair to a list
}

void synchronize_rcu(void)
{
        int cpu, ncpus = 0;

        for_each_cpu(cpu)
                schedule_current_task_to(cpu);

        for each entry in the call_rcu list
                entry->callback (entry->arg);
}
```

In the code sample, `rcu_assign_pointer` and `rcu_dereference` can be ignored without missing much. However, they are provided below.

```
#define rcu_assign_pointer(p, v)        ({ \
                                                smp_wmb(); \
                                                ACCESS_ONCE(p) = (v); \
                                        })

#define rcu_dereference(p)              ({ \
                                                typeof(p) _value =
ACCESS_ONCE(p); \
smp_read_barrier_depends(); /* nop on most architectures */ \
                                                (_value); \
                                        })
```

Note that `rcu_read_lock` and `rcu_read_unlock` do absolutely nothing. This is the great strength of classic RCU in a non-preemptive kernel: read-side overhead is precisely zero, as `smp_read_barrier_depends()` is an empty macro on all but DEC Alpha CPUs;[19] such memory barriers are not needed on modern CPUs. The `ACCESS_ONCE()` macro is a volatile cast that generates no additional code in most cases. And there is absolutely no way that `rcu_read_lock` can participate in a deadlock cycle, cause a realtime process to miss its scheduling deadline, precipitate priority inversion, or result in high lock contention. However, in this toy RCU implementation, blocking within an RCU read-side critical section is illegal, just as is blocking while holding a pure spinlock.

The implementation of `synchronize_rcu` moves the caller of synchronize_cpu to each CPU, thus blocking until all CPUs have been able to perform the context switch. Recall that this is a non-preemptive environment and that blocking within an RCU read-side critical section is illegal, which imply that there can be no preemption points within an RCU read-side critical section. Therefore, if a given CPU executes a context switch (to schedule another process), we know that this CPU must have completed all preceding RCU read-side critical sections. Once all CPUs have executed a context switch, then all preceding RCU read-side critical sections will have completed.

Analogy with reader-writer locking [edit]

Although RCU can be used in many different ways, a very common use of RCU is analogous to reader-writer locking. The following side-by-side code display shows how closely related reader-writer locking (on the left) and RCU (on the right) can be.[20]

```c
struct el {
  struct list_head lp;
  long key;
  spinlock_t mutex;
  int data;
  /* Other data fields */
};
DEFINE_RWLOCK(listmutex);
LIST_HEAD(head);
```

```c
struct el {
  struct list_head lp;
  long key;
  spinlock_t mutex;
  int data;
  /* Other data fields */
};
DEFINE_SPINLOCK(listmutex);
LIST_HEAD(head);
```

```c
int search(long key, int *result)
{
  struct el *p;

  read_lock(&listmutex);
  list_for_each_entry(p, &head, lp) {
    if (p->key == key) {
      *result = p->data;
      read_unlock(&listmutex);
      return 1;
    }
  }
  read_unlock(&listmutex);
  return 0;
}
```

```c
int search(long key, int *result)
{
  struct el *p;

  rcu_read_lock();
  list_for_each_entry_rcu(p, &head, lp) {
    if (p->key == key) {
      *result = p->data;
      rcu_read_unlock();
      return 1;
    }
  }
  rcu_read_unlock();
  return 0;
}
```

```c
int delete(long key)
{
  struct el *p;

  write_lock(&listmutex);
  list_for_each_entry(p, &head, lp) {
    if (p->key == key) {
      list_del(&p->lp);
      write_unlock(&listmutex);
      kfree(p);
      return 1;
    }
  }
  write_unlock(&listmutex);
  return 0;
}
```

```c
int delete(long key)
{
  struct el *p;

  spin_lock(&listmutex);
  list_for_each_entry(p,
    if (p->key == key) {
      list_del_rcu(&p->lp);
      spin_unlock(&listmutex);
      synchronize_rcu();
      kfree(p);
      return 1;
    }
  }
  spin_unlock(&listmutex);
  return 0;
}
```

The differences between the two approaches are quite small. Read-side locking moves to `rcu_read_lock` and `rcu_read_unlock`, update-side locking moves from a reader-writer lock to a simple spinlock, and a `synchronize_rcu` precedes the `kfree`.

However, there is one potential catch: the read-side and update-side critical sections can now run concurrently. In many cases, this will not be a problem, but it is necessary to check carefully regardless. For example, if multiple independent list updates must be seen as a single atomic update, converting to RCU will require special care.

Also, the presence of `synchronize_rcu` means that the RCU version of `delete` can now block. If this is a problem, `call_rcu` could be used like `call_rcu (kfree, p)` in place of `synchronize_rcu`. This is especially useful in combination with reference counting.

Name [edit]

The name comes from the way that RCU is used to update a linked structure in place. A thread wishing to do this uses the following steps:

- create a new structure,
- copy the data from the old structure into the new one, and save a pointer to the old structure,
- modify the new, copied, structure
- update the global pointer to refer to the new structure, and then
- sleep until the operating system kernel determines that there are no readers left using the old structure, for example, in the Linux kernel, by using synchronize_rcu().

When the thread which made the copy is awakened by the kernel, it can safely deallocate the old structure.

So the structure is *read* concurrently with a thread *copying* in order to do an *update*, hence the name "read-copy update". The abbreviation "RCU" was one of many contributions by the Linux community. Other names for similar techniques include *passive serialization* and *MP defer* by VM/XA programmers and *generations* by K42 and Tornado[*dead link*] programmers.

History [edit]

This section **is in a list format that may be better presented using prose.** You can help by converting this section to prose, if appropriate. Editing help is available. *(May 2014)*

Techniques and mechanisms resembling RCU have been independently invented multiple times:[21]

1. H. T. Kung and Q. Lehman described use of garbage collectors to implement RCU-like access to a binary search tree.[22]
2. Udi Manber and Richard Ladner extended Kung's and Lehman's work to non-garbage-collected environments by deferring reclamation until all threads running at removal time have terminated, which works in environments that do not have long-lived threads.[23]
3. Richard Rashid et al. described a lazy translation lookaside buffer (TLB) implementation that deferred reclaiming virtual-address space until all CPUs flushed their TLB, which is similar in spirit to some RCU implementations.[24]
4. James P. Hennessy, Damian L. Osisek, and Joseph W. Seigh, II were granted US Patent 4,809,168 in 1989 (since lapsed). This patent describes an RCU-like mechanism that was apparently used in VM/XA on IBM mainframes.[25]
5. William Pugh described an RCU-like mechanism that relied on explicit flag-setting by readers.[26]
6. Aju John proposed an RCU-like implementation where updaters simply wait for a fixed period of time, under the assumption that readers would all complete within that fixed time, as might be appropriate in a hard real-time system.[27] Van Jacobson proposed a similar scheme in 1993 (verbal communication).
7. J. Slingwine and P. E. McKenney received US Patent 5,442,758 in August 1995, which describes RCU as implemented in DYNIX/ptx and later in the Linux kernel.[28]
8. B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm described an RCU-like mechanism used in the University of Toronto Tornado research operating system[*dead link*] and the closely related IBM Research K42 research operating systems.[29]
9. Rusty Russell and Phil Rumpf described RCU-like techniques for handling unloading of

Linux kernel modules.[30][31]

10. D. Sarma added RCU to version 2.5.43 of the Linux kernel in October 2002.
11. Robert Colvin et al. formally verified a lazy concurrent list-based set algorithm that resembles RCU.[32]
12. M. Desnoyers et al. published a description of user-space RCU.[33][34]
13. A. Gotsman et al. derived formal semantics for RCU based on separation logic.[35]
14. Ilan Frenkel, Roman Geller, Yoram Ramberg, and Yoram Snir were granted US Patent 7,099,932 in 2006. This patent describes an RCU-like mechanism for retrieving and storing quality of service policy management information using a directory service in a manner that enforces read/write consistency and enables read/write concurrency.[36]

See also [edit]

- Concurrency control
- Multiversion concurrency control
- Resource contention
- Lock (software engineering)
- Lock-free and wait-free algorithms
- Pre-emptive multitasking
- Real-time computing
- Resource starvation
- Synchronization

Notes [edit]

1. ^ RCU does not implement mutual exclusion in the conventional sense: RCU readers can and do run

concurrently with RCU updates. RCU's variant of mutual exclusion is in space, with RCU readers accessing old versions of data being concurrently updated, rather than in time, as is the case for conventional concurrency-control mechanisms.

2. ^ Only readers that are active during the removal phase need be considered, because any reader starting after the removal phase will be unable to gain a reference to the removed data items, and therefore cannot be disrupted by the reclamation phase.
3. ^ Garbage collectors, where available, may be used to perform this step.
4. ^ RCU-based deadlocks are still possible, for example by executing a statement that blocks until a grace period completes within an RCU read-side critical section.

References [edit]

1. ^ Guniguntala, Dinakar; McKenney, Paul E.; Triplett, Joshua; Walpole, Jonathan (April–June 2008). "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux". *IBM Systems Journal* **47** (2): 221–236. doi:10.1147/sj.472.0221.
2. ^ McKenney, Paul E.; Walpole, Jonathan (December 17, 2007). "What is RCU, Fundamentally?". Linux Weekly News. Retrieved September 24, 2010.
3. ^ McKenney, Paul E.; Slingwine, John D. (October 1998). *Read-Copy Update: Using Execution History to Solve Concurrency Problems* (PDF). *Parallel and Distributed Computing and Systems*: 509–518. External link in |work= (help)
4. ^ McKenney, Paul E.; Walpole, Jonathan (July 2008). "Introducing technology into the Linux kernel: a case study". *SIGOPS Oper. Syst. Rev.* **42** (5): 4–17. doi:10.1145/1400097.1400099.
5. ^ Olsson, Robert; Nilsson, Stefan (May 2007). *TRASH: A dynamic LC-trie and hash table structure*. *Workshop on High Performance Switching and Routing (HPSR'07)*.
6. ^ Piggin, Nick (July 2006). *A Lockless Pagecache in Linux---Introduction, Progress, Performance*. *Ottawa Linux Symposium*.
7. ^ http://www.rdrop.com/users/paulmck/RCU/linuxusage.html

8. ^ Kannan, Hari (2009). *Ordering decoupled metadata accesses in multiprocessors*. *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA): 381–390. doi:10.1145/1669112.1669161. ISBN 978-1-60558-798-1.
9. ^ Matthews, Chris; Coady, Yvonne; Appavoo, Jonathan (2009). *Portability events: a programming model for scalable system infrastructures*. *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems* (San Jose, CA, USA). doi:10.1145/1215995.1216006. ISBN 1-59593-577-0.
10. ^ Da Silva, Dilma; Krieger, Orran; Wisniewski, Robert W.; Waterland, Amos; Tam, David; Baumann, Andrew (April 2006). "K42: an infrastructure for operating system research". *SIGOPS Oper. Syst. Rev.* **40** (2): 34–42. doi:10.1145/1131322.1131333.
11. ^ Appavoo, Jonathan; Da Silva, Dilma; Krieger, Orran; Auslander, Mark; Ostrowski, Michal; Rosenburg, Bryan; Waterland, Amos; Wisniewski, Robert W.; Xenidis, Jimi (August 2007). "Experience distributing objects in an SMMP OS". *ACM Trans. Comput. Syst.* **25** (3): 6/1–6/52. doi:10.1145/1275517.1275518.
12. ^ Fraser, Keir; Harris, Tim (2007). "Concurrent programming without locks". *ACM Trans. Comput. Syst.* (New York, NY, USA: ACM) **25** (2): 34–42. doi:10.1145/1233307.1233309.
13. ^ Porter, Donale E.; Hofmann, Owen S.; Rossbach, Christopher J.; Benn, Alexander; Witchel, Emmett (2009). *Operating systems transactionsinfrastructures*. *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (Big Sky, MT, USA: ACM). doi:10.1145/1629575.1629591. ISBN 978-1-60558-752-3.
14. ^ Hart, Thomas E.; McKenney, Paul E.; Demke Brown, Angela; Walpole, Jonathan (December 2007). "Performance of memory reclamation for lockless synchronization". *J. Parallel Distrib. Comput.* **67**: 1270–1285. doi:10.1016/j.jpdc.2007.04.010.
15. ^ McKenney, Paul E. (January 4, 2008). "RCU part 2: Usage". Linux Weekly News. Retrieved September 24, 2010.
16. ^ Desnoyers, Mathieu (December 2009). *Low-Impact Operating System Tracing* (PDF). *Ecole Polytechnique de Montreal* (Thesis).
17. ^ McKenney, Paul E. (January 17, 2008). "RCU part 3: the RCU API". Linux Weekly News.

Retrieved September 24, 2010.

18. ^ McKenney, Paul E.; Appavoo, Jonathan; Kleen, Andi; Krieger, Orran; Russell, Rusty; Sarma, Dipankar; Soni, Maneesh (July 2001). *Read-Copy Update* (PDF). *Ottawa Linux Symposium*.
19. ^ Wizard, The (August 2001). "Shared Memory, Threads, Interprocess Communication". Hewlett-Packard. Retrieved December 26, 2010.
20. ^ McKenney, Paul E. (October 2003). "Using {RCU} in the {Linux} 2.5 Kernel". Linux Journal. Retrieved September 24, 2010.
21. ^ McKenney, Paul E. (July 2004). *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques* (PDF). *OGI School of Science and Engineering at Oregon Health and Sciences University* (Thesis).
22. ^ Kung, H. T.; Lehman, Q. (September 1980). "Concurrent Maintenance of Binary Search Trees". *ACM Transactions on Database Systems* **5** (3): 354. doi:10.1145/320613.320619.
23. ^ Manber, Udi; Ladner, Richard E. (September 1984). "Concurrency Control in a Dynamic Search Structure". *ACM Transactions on Database Systems* **9** (3).
24. ^ Rashid, Richard; Tevanian, Avadis; Young, Michael; Golub, David; Baron, Robert; Bolosky, William; Chew, Jonathan (October 1987). *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures* (PDF). *Second Symposium on Architectural Support for Programming Languages and Operating Systems* (Association for Computing Machinery).
25. ^ US 4809168, "Passive Serialization in a Multitasking Environment"
26. ^ Pugh, William (June 1990). *Concurrent Maintenance of Skip Lists* (Technical report). Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland. CS-TR-2222.1.
27. ^ John, Aju (January 1995). *Dynamic vnodes — design and implementation*. *USENIX Winter 1995*.
28. ^ US 5442758, "Apparatus and Method for Achieving Reduced Overhead Mutual Exclusion and Maintaining Coherency in a Multiprocessor System"
29. ^ Gamsa, Ben; Krieger, Orran; Appavoo, Jonathan; Stumm, Michael (February 1999). *Tornado:*

Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System (PDF). *Proceedings of the Third Symposium on Operating System Design and Implementation*.

30. ^ Russell, Rusty (June 2000). "Re: modular net drivers".
31. ^ Russell, Rusty (June 2000). "Re: modular net drivers".
32. ^ Colvin, Robert; Groves, Lindsay; Luchangco, Victor; Moir, Mark06 (August 2006). *Formal Verification of a Lazy Concurrent List-Based Set Algorithm* (PDF). *Computer Aided Verification (CAV 2006)*.
33. ^ Desnoyers, Mathieu; McKenney, Paul E.; Stern, Alan; Dagenais, Michel R.; Walpole, Jonathan (February 2012). *User-Level Implementations of Read-Copy Update*. *IEEE Transactions on Parallel and Distributed Systems*.
34. ^ McKenney, Paul E.; Desnoyers, Mathieu; Jiangshan, Lai (November 13, 2013). "User-space RCU". Linux Weekly News. Retrieved November 17, 2013.
35. ^ Gotsman, Alexey; Rinetzky, Noam; Yang, Hongseok (March 16–24, 2013). *Verifying concurrent memory reclamation algorithms with grace* (PDF). *ESOP'13: European Symposium on Programming*.
36. ^ US 7099932, Frenkel, Ilan; Geller, Roman & Ramberg, Yoram et al., "Method and apparatus for retrieving network quality of service policy information from a directory in a quality of service policy management system", published 2006-08-29, assigned to Cisco Tech Inc.

Bauer, R.T., (June 2009), "Operational Verification of a Relativistic Program" PSU Tech Report TR-09-04 (http://www.pdx.edu/sites/www.pdx.edu.computer-science/files/tr0904.pdf)

External links [edit]

- Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan: User-space RCU. *Linux Weekly News.*
- Paul E. McKenney and Jonathan Walpole: What is RCU, Fundamentally?, What is RCU? Part 2: Usage, and RCU part 3: the RCU API. *Linux Weekly News.*

- Paul E. McKenney's RCU web page
- Hart, McKenney, and Demke Brown (2006). *Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation* An IPDPS 2006 Best Paper comparing RCU's performance to that of other lockless synchronization mechanisms. *Journal version* (including Walpole as author).
- U.S. Patent 5,442,758 (1995) "Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring"
- Paul McKenney: Sleepable RCU. *Linux Weekly News.*

Categories: Operating system technology | Concurrency control